# Customization, reusability and near-miss design

Martin S. Feather
USC / Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
U. S. A.
feather@isi.edu

*In 'Automating Software Design: Interactive Design' Workshop Notes from the Ninth National Conference on Artificial Intelligence (AAAI-91), pages 34-39. Workshop Notes available as Technical Report No. RS-91-287 from Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292, USA.*

## 1 Introduction

Reusable software components offer the promise of facilitating the rapid construction of large and complex software systems. One of the impediments to the success of this approach, however, is the tension between making reusable components as widely applicable as possible, while retaining their individual simplicity and effectiveness. Wide applicability of individual components is desirable so that they can be reused in many circumstances, thus avoiding having to populate such a library with many similar versions of each component. Conversely, the more options and features that are built in to a single component, the less simple it is to construct in the first place, and the more work is required to correctly instantiate it for some given use.

'Evolution transformations' are suggested as a means to alleviate this dilemma. Applied to a software component, an evolution transformation changes the meaning of that component in some well-determined manner. This promises to make a library of software components more versatile it is no longer necessary to retrieve a component that exactly matches the case in hand; rather, a component close to the need would suffice, after which evolution transformations would be applied to customize it appropriately. This means the library can be smaller, and individual components can be simpler. Furthermore, the organization of components in the library can profitably use these evolution transformations as the relationships between pairs of closely related components, neither of which is a strict refinement of the other.

What does this have to do with interactive design? If brings to the fore a mode of interaction concerned with mismatch and evolution, rather than with selection and instantiation. If this is so, it is appropriate to consider how interaction can support this activity.

These ideas are, to date, merely ideas. Within the ARIES project (of ISI and Lockheed Sanders [7]), we have developed and used evolution transformations to help in requirements acquisition and specification construction [6, 5]. However, we have not (yet) investigated the above hypotheses for software component libraries. Nevertheless, it is on the basis of the insights gained from our experience with evolution transformations that my above proposals are formulated. The next section offers a simple example of specification and evolution as brief illustration of the above points. In the final section I make some observations on the implications of this approach for interactive design, relating these general observations to specific points of the example.

## 2 Example of Specification and Evolution

The example will present a simple initial specification, two different evolutions of that specification, and finally the specification that combines both of those evolutions. Two points should be borne in mind while viewing this example:

- The example deals only with specifications of a task, not the programs that would actually implement those specifications. This is a consequence of our particular view of requirements, specification and implementation. We look to program transformation to bridge the gap between specification and implementation. Preliminary results suggest that the transformational development of a specification can be reused to good effect to facilitate the transformational development of an changed (evolved) version of that specification [4, 3]. Furthermore, we believe that for most real-world problems the development of a formal specification is itself a difficult activity, in many cases more difficult than the subsequent implementation from that specification. Thus, since transformational implementation is not the focus of this paper, we deal only with specifications and their evolutions.

- The simple initial specification that begins the example might seem too simple. Such simplicity is deliberate, however, because it allows us to start from an idealized abstraction of the problem, and thereafter incrementally and independently introduce the details that characterize more realistic versions of the specification.

### 2.1 Evolution Example - Retail Company

The example concerns inventory control within a retail company, and is loosely based on the specification given in [9]. Briefly, this involves a stock of parts, customers ordering parts, and a factory restocking parts when the company's stock is exhausted. It is easy to imagine that a library of software specification components would contain descriptions of systems of this nature.

Since the primary goal is to illustrate the issues of near-miss specification and customization, I will not provide descriptions of the specification language or evolution transformations beyond those necessary to motivate this example. More details of the specification language, Gist, and the role of evolution transformations in the parallel elaboration of specifications can be found in [2].

### 2.2 Simple initial specification

We begin with the specification of an extremely simple instance of such a company and its operating environment (namely, customers that issue orders, and factory that replenishes parts). This captures the spirit of the operations of a retail company, but avoids many details. An explanation of the concepts used follows the specification.

```
type part;

relation Owner(part,
               owner:customer union retail-company union factory) ;

type order;

relation Customer(part.customer) };

type customer with
{ demon ISSUE-ORDER() := create order with
       { Customer <- (the customer) } };
```

```
type retail-company with
{ demon HANDLE-ORDER(order)
  when start extant order
  := let ( p:part | p.Owner = (the retail-company) )
      p.Owner <- order.Customer;

demon REPLENISH()
when start not exists (p:part)
          p.Owner = (the retail-company)
:= ASK-FOR-REPLENISHMENT() };

type factory with
{ var replenishment-quantity:integer;
procedure ASK-FOR-REPLENISHMENT()
:= loop for x from 1 to replenishment-quantity
   do create part with { Owner <- (the retail-company) } }
```

Types and relations model information in the retail-company world: `part`, `order`, `customer`, `retail-company` and `factory` are declared as types; objects in the specification will be instances of these. `Customer` and `Owner` are declared as binary relations - the former relates objects of type `order` to objects of type `customer`, while the latter relates objects of type `part` to objects of the union of the types `customer`, `retail-company` and `factory`.

Relations queries retrieve objects associated with other objects, e.g., `order.Customer` retrieves the customer object that is associated to order by the `Customer` relation.

Execution of statements changes this information by creating or destroying objects, and by inserting or removing relationships among objects. E.g., `p. Owner <- order.Customer` changes the `Owner` relationship between part `p` and its current owner to instead relate `p` to the customer object denoted by `order.Customer` - i.e., this statement models the transfer of a part to the customer who issued the order (and furthermore, because the part was selected to be one whose owner was the retail company - `p:part | p.Owner = (the retail-company)` - the transferred part must come from the retail-company. Object creation is illustrated in `create order with { Customer <- (the customer) }`, which creates a new object of type `order` and puts it into the `Customer` relation with the object denoted by the `customer` (because of the context in which this occurs, this is the customer who is creating the order).

Procedures provide the usual procedural abstraction. Here `ASK-FOR-REPLENISHMENT` is used to communicate between the retail-company and the factory.

Demons cause activity to happen in response to situations arising. E.g., `HANDLE-ORDER` has the keyword when followed by the condition (a predicate) start extant order. When this predicate is true (i.e., when the order first comes into being), the demon initiates its response (the statement following the `:=` symbol). In this case, the response is the activity to transfer a part to the customer. Omitting the when and its condition is equivalent to nondeterministically choosing to perform the response (as in the `ISSUE-ORDER` demon, modelling customers unpredictably issuing orders to be filled by the retail company). The instance of the type to which a demon belongs is the object which is said to be performing the response, e.g., the retail-company performs the response of the `HANDLE-ORDER` demon, individual customers perform the response of the `ISSUE-ORDER` demon, etc.

There are a number of obvious ways in which the above is trivial, e.g., there is only one kind of part; an order is always for a single part. The purpose is to capture the 'essence' of the retail-company notion in this specification. We examine next how to use evolution transformations to introduce more details, exceptional cases, etc.

2.3 Evolving the initial specification to have multiple kinds of parts

As the first example of evolution, consider that there could be different kinds of parts. Thus, customers' orders must additionally specify the kind of part, the `HANDLE-ORDER` demon must give the customer the appropriate kind of part, and the `REPLENISH` demon must ask for replenishment of the appropriate kind of part when the company stocks of that kind are exhausted. Essentially, we must parameterize the specification where it deals with parts to take into account the kind of part. The end result of this is the following slightly more complicated specification (new or changed lines are marked with `|` at the left):

```
| type kind;

  type part;

  relation Owner(part,
                 owner:customer union retail-company union factory) ;

| relation Part-Kind(part,kind);

  type order;

  relation Customer(part,customer);

| relation Order-Kind(order,kind);

  type customer with
  { demon ORDER()
    := create order with
       { Customer <- (the customer),
|        Order-Kind <- (any kind) } };

  type retail-company with
  { demon HANDLE-ORDER(order)
    when start extant order
    := let ( p:part |
             p.Owner = (the retail-company)
|            and
|            p.Part-Kind = order.Order-Kind )
       p.Owner <- order.Customer;

| demon REPLENISH(k:kind)
| when start not exists (p:part)
|         p.Owner = (the retail-company) and
|         p.Part-Kind = k
| := ASK-FOR-REPLENISHMENT(k) };
  type factory with
  { var replenishment-quantity:integer;

| procedure ASK-FOR-REPLENISHMENT(k:kind)
  := loop for x from 1 to replenishment-quantity
     do create part with
```

```
        { Owner <- (the retail-company),
|         Part-Kind <- k } }
```

When an order is created, in addition to inserting it into the `Customer` relation with the customer creating that order (`Customer <- (the customer)`, as before), it is also inserted into the new `Order-Kind` relation with some non-deterministically chosen kind (`Order-Kind <- (any kind)`). When an order is handled, the part transferred to the customer must also be of the appropriate kind (ensured by `p.Part-Kind = order.Order-Kind`). Finally, when replenishment occurs, it is for a particular kind of `part`. Note however that the replenishment quantity has not been parameterized by the kind of part being replenished, although it certainly could have been. There are obviously numerous variations on how the parameterization is actually performed, each of which would give rise to a slightly different specification.

### 2.4 Evolving the initial specification to have quantities in orders

In a different evolution, we introduce the notion that each order has a non-zero quantity. Note that this is an evolution of the original specification, not of the specification evolved to have multiple kinds of parts. This is to separate concerns, and retain simplicity.

```
  type part;

  relation Ovner(part,
                 owner:customer union retail-company union factory);

  type order;

  relation Customer(part,customer);

| relation Quantity(order,integer)

  type customer with
  { demon ISSUE-ORDER()
    := create order with
       { Customer <- (the customer),
         Quantity <- (any i:integer | i>0) } };

  type retail-company with
  { demon HANDLE-ORDER(order) \
    when start extant order
|   := let ( q:integer | order.Quantity = q,
|             setp:setof(part) |
|               size(setp)= q and
|            for-all (p:part) p in setp => p.Owner = (the retail-company) ) |  |
|      atomic (loop (p over setp)
|               do p.Owner <- order.Customer);

  demon REPLENISH()
  when start not exists (p:part)
            p.Owner = (the retail-company)
  := ASK-FOR-REPLENISHMENT() };

  type factory with
  { var replenishment-quantity:integer;

  procedure ASK-FOR-REPLENISHMENT()
```

```
  := loop for x from 1 to replenishment-quantity
     do create part with
        { Owner <- (the retail-company) } }
```

The more complex form of the statement of `HANDLE-ORDER` essentially selects a set of parts such that the size of that set is the quantity of the order (`size(setp) = q`) and each element of that set is a part owned by the retail-company (`for-all (p:part) ...`).

There may be a problem with the above specification if a customer orders a quantity of parts greater than the company's stock, in which case the order will not be handled until the level of stock is replenished to (at least) the quantity of the order (which need not necessarily occur if the replenishment quantity is small relative to the order). This could be dealt with by having `HANDLE-ORDER` divide large orders into two, so that the first portion can be filled immediately, while the remainder is put on back-order, or alternatively having pending orders cause a request for an appropriate quantity of replenishment. We will not pursue either of these options here, but clearly any reasonable codification of this knowledge would either add these as further evolutions, or at least keep some record of the potential inadequacies of this specification.

2.5 Combining evolutions

Having evolved the specification in two different ways, we now consider how to bring those together, i.e., how to achieve a single specification with part kinds and order quantities. The advantage of this approach is that it encourages the explicit consideration of the interactions that arise during combination.

In combining our two evolutions of the initial specification of the retail company, we most likely would wish to allow customers to issue orders which can have different quantities for various kinds of parts, and have the retail company handle such orders accordingly. We may need to take a moment to consider how the requirement on the quantity of an order being non-zero (this requirement is embedded in the `ISSUE-ORDER` demon) is to be evolved. Presumably, we wish to insist only that the total quantity of parts ordered is non-zero, not that the number of parts of each kind be non-zero. Other than this detail, combination is relatively straightforward, although leads to a yet-larger specification than the initial specification, or either of its evolutions (as should be expected).

The specification that results is shown below - each order is related to some number of suborders, each of which is related to a kind and a quantity. When an order is created, a suborder for some non-zero quantity of each kind of part is *optionally* created, subject to the postcondition that there must be a suborder for at least one kind of part (thus ensuring that the total quantity of parts ordered is non-zero, without requiring that the number of parts of each kind is non-zero, as discussed above). When an order is handled, the appropriate set of parts of appropriate kinds is transferred to the customer.

```
type kind;

type part;

relation Owner(part,
               owner:customer union retail-company union factory);

relation Part-Kind(part,kind);
```

```
type order;

type suborder;

relation Customer(part,customer);

relation SubOrderOf(suborder,order);

relation Order-Kind(suborder,kind);

relation Quantity(suborder,integer);

type customer with
{ demon ISSUE-ORDER()
  := atomic
     (create order with
        { Customer <- (the customer) },
      loop (k | kind)
      do optionally
         create suborder with
         { SubOrderOf <- order;
           Order-Kind <- k;
           Quantity <- (any i:integer | i>0) }
     postcondition exists (s:suborder) SubOrderOf(s,order) ) };

type retail-company with
{ demon HANDLE-ORDER(order)
  when start extant order
  := loop (s | SubOrderOf(s,order) )
     do let ( q:integer | s.Quantity = q,
              setp:setof(part) |
                 size(setp)= q and
                 for-all (p:part) p in setp =>
                     p.Owner = (the retail-company) and
                     p.Part-Kind = s.Order-Kind )
        atomic (loop (p over setp) do p.Owner <- order.Customer);

demon REPLENISH(k:kind)
when start not exists (p:part)
        p.Owner = (the retail-company) and p.Part-Kind = k
:= ASK-FOR-REPLENISHMENT(k) };

type factory with
{ var replenishment-quantity:integer;

procedure ASK-FOR-REPLENISHMENT(k:kind)
:= loop for x from 1 to replenishment-quantity
   do create part with
      { Owner <- (the retail-company),
        Part-Kind <- k } }
```

## 3 Implications for interactive design

### 3.1 Libraries, instantiation and evolution

In the retail company example, extra details introduced in both of the evolutions added to the size and complexity of the specification. Furthermore, even for the simple evolutions of this example,

further options were apparent (e.g., whether or not to parameterize the replenishment quantity by the part kind). In general we may expect a single specification that encompassed a broad range of retail company possibilities to be rather large and complex. Alternatively, providing coverage (in a library of components) by populating that library with a multitude of versions of retail company like specifications would lead to a library of unwieldy size. Our hope is that evolution offers a third alternative, in which the library's content of components is complemented by an array of evolution transformations. The user would select a component near to the one wanted, and then use evolution transformations to customize that component. Because similar evolutions can be applied to many different components (e.g., parameterization is a very general concept), they effectively multiply the size of the library. This is at the cost of requiring a different form of interaction with the user, one involving sequences of evolution rather than the traditional instantiation of a generic component.

### 3.2 Refinement, evolution and designing from ideals

The incremental style of development (of programs, specifications or requirements) in which an initial very abstract formulation of some task is incrementally refined to introduce successive levels of detail has been well-studied and has advantages. Our evolution transformations are not limited to pure refinement; in the retail company example, the introduction of quantities into orders and their handling gave rise to new possibilities of 'starvation' (large orders that remain unfilled) that are not readily seen as refinements of the behaviors of the initial specification. Evolutions thus offer an extension of the paradigm. One consequence ofthis is that it is possible to start with much more idealized statements of specifications, relying upon evolution to retract the unrealistic assumptions. The development history then records how thos~ ideals were compromised. This history is useful for both clear exposition and future maintenance: if changes occur to the environment surrounding the software, then it may be possible to look back through the development history and see how to adapt to those changed conditions. Without such a history, that is, left with only the end-product of a design process, we would be forced to rediscover design decisions in the face of change.

### 3.3 Knowledge organization and composition

It is clear that other approaches to structuring and manipulating knowledge organized as software components will continue to have application - evolution transformations should supplement, not supplant them. *Composition* of components is one such approach (e.g., ACT TWO [1]). In the retail company example no mention was made of scheduling the handling of customer orders the specification denoted all possible schedulings, without imposing any criteria such as handling orders in the order in which they were received. I consider scheduling to be a basic notion which has its own space of elaborations (structured using refinements and evolutions, of course) to provide alternative scheduling strategies (e.g., FIFO, priority levels) and elaborated behaviors (e.g., putting requests on hold, cancellation, preempting). A realistic retail company specification will compose ordering and replenishment activity with scheduling concepts and mechanisms. Organization of these classes of components (scheduling, inventory control, etc) may well follow a more traditional hierarchical form such as that suggested by the Requirements Apprentice's cliché library [8].

### 3.4 Near-miss specification and mismatch discovery

If near-miss specification using evolution and customization is to be put into practice, it requires support not only for conducting evolution but also for detecting the need for evolution. It would no longer be the case that an 'incorrect' specification (or piece of specification) arose rarely and only

by mistake, rather it would be the norm to expect to progress through an alternation of mismatch-detection and evolution. Comparisons between the state of the design so far and other descriptions that the user provides (e.g., scenarios of system behavior, concrete examples, specific viewpoints) would become the focus of the design activity, and comprise a major part of communication between system and user. Selection of the appropriate evolution is one of the key steps of this proposed methodology, and should be a cooperative activity blending machine capabilities and human responsibilities. Human input is required to indicate deficiencies in the current form of the specification, or provide further information (the other descriptions referred to above) which the system can detect deficiencies. The machine can then search for all the evolutions that would potentially correct these deficiencies. Finally the human picks out the particular evolution(s) to be applied. In our ARIES system we employ this strategy to pick the evolution transformations to be used for specification construction [5], and it seems reasonable to suppose that the same strategy will apply to customization of specifications.

## 4 Acknowledgments

This research has been supported in part by Defense Advanced Research Projects Agency grant No. NCC-2-520, and in part by Rome Air Development Center contract No. F30602-85-C-0221 and F30602-89-C-0103. Views and conclusions contained in this document are those of the author and should not be interpreted as representing the official opinion or policy of DARPA, RADC, NSF, the U.S. Government, or any other person or agency connected with them. The author wishes to acknowledge the Software Sciences Division at ISI for providing the context in which this research has been conducted.

## References

[1] H. Ehrig, P. Boehm, and W. Fey. Algebraic concepts for formal specifications and transformation of modular software systems. Technical Report Bericht-Nr. 90/4, Technische Universitaet Berlin, 1990.

[2] M.S. Feather. Constructing specifications by combining parallel elaborations. *IEEE Transactions on Software Engineering*, 15(2):198-208, February 1989.

[3] M.S. Feather. Specification evolution and program (re)transformation. In *Proceedings of the 5th Annual RADC Knowledge-Based Software Assistant (KBSA) Conference*, Liverpool, NY, September 1990, pages 403-417, 1990.

[4] A. Goldberg. Reusing software developments. In *Proceedings, 4th Annual Knowledge-Based Software Assistant (KBSA) Conference*, 1989.

[5] W.L. Johnson and M.S. Feather. Building an evolution transformation library. In *Proceedings, 12th International Conference on Software Engineering*, Nice, France, pages 238-248. IEEE Computer Society Press, March 1990.

[6] W.L. Johnson and M.S. Feather. Using evolution transformations to construct specifications. In M. Lowry and R. McCartney, editors, *Automating Software Design*. AAAI Press, 1991.

[7] W.L. Johnson and D.R. Harris. Requirements analysis using aries: themes and examples. In *Proceedings of the 5th Annual RADC Knowledge-Based Software Assistant (KBSA) Conference*, Liverpool, NY, September 1990, pages 121-131,1990.

[8] H.B. Reubenstein and R.C. Waters. The requirements apprentice: An initial scenario. In *Proceedings, 5th International Workshop on Software Specification and Design*, Pittsburgh, Pennsylvania, USA, pages 211-218. Computer Society Press of the IEEE, 1989.

[9] A. Solvberg and C.H. Kung. On structural and behavioral modelling of reality. In *Proceedings of the IPIP TC2 WG2.5 WC on Database Semantics*, Hasselt, Belgium, 1985.